VisualWorks®

Advanced Tools

User's Guide

ObjectShare®

Non-Commercial

Part Number: Non-Commercial

Software Release 3.0

This document is subject to change without notice.

RESTRICTED RIGHTS LEGEND:

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

Trademark acknowledgments:

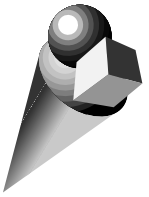
ParcPlace and VisualWorks are trademarks of ObjectShare, Inc., its subsidiaries, or successors and are registered in the United States and other countries. ObjectLens, ObjectSupport, ParcPlace Smalltalk, Database Connect, DLL & C Connect, and COM Connect are trademarks of ObjectShare, Inc., its subsidiaries, or successors. ENVY is a registered trademark of Object Technology International, Inc. All other products or services mentioned herein are trademarks of their respective companies. Specifications subject to change without notice.

The following copyright notices apply to software that accompanies this documentation:

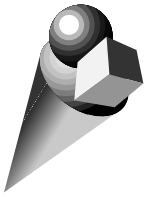
VisualWorks is furnished under a license and may not be used, copied, disclosed, and/or distributed except in accordance with the terms of said license. No class names, hierarchies, or protocols may be copied for implementation in other systems.

This manual set and online system documentation copyright © 1995–98 by ObjectShare, Inc. All rights reserved. No part of it may be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior written consent from ObjectShare.

ObjectShare, Inc., 16811 Hale Avenue, Irvine, California 92606-5089



Contents



About This Book

The *Advanced Tools User's Guide* provides the experienced VisualWorks® developer with information necessary to use the tools and reusable code provided with this add-on component to VisualWorks.

Conventions

We have followed a variety of conventions, which are standard in the VisualWorks documentation.

Typographic Conventions

The following fonts are used to indicate special terms:

Example	Description
template	Indicates new terms where they are defined, emphasized words, book titles, and words as words.
`cover.doc`	Indicates filenames, pathnames, commands, and other constructs to be entered outside VisualWorks (for example, at a command line).
`filename.xwd`	Indicates a variable element for which you must substitute a value.
windowSpec	Indicates Smalltalk constructs; it also indicates any other information that you enter through the VisualWorks graphical user interface.
Edit menu	Indicates VisualWorks user-interface labels for menu names, dialog-box fields, and buttons; it also indicates emphasis in Smalltalk code samples.

Special Symbols

This book uses the following symbols to designate certain items or relationships:

Examples	Description
File→New command	Indicates the name of an item (New) on a menu (File).
<Return> key <Select> button <Operate> menu	Indicates the name of a keyboard key or mouse button; it also indicates the pop-up menu that is displayed by pressing the mouse button of the same name.
<Control>-<g>	Indicates two keys that must be pressed simultaneously.
<Escape> <c>	Indicates two keys that must be pressed sequentially.
Integer>>asCharacter	Indicates an instance method defined in a class.
Float class>>pi	Indicates a class method defined in a class.

Mouse Buttons and Menus

VisualWorks supports a one-, two-, or three-button mouse common on various platforms. Smalltalk traditionally expects a three-button mouse, which we denote by the logical names <Select>, <Operate>, and <Window>:

<Select> button	*Select* (or choose) a window location or a menu item, position the text cursor, or highlight text.
<Operate> button	Bring up a menu of *operations* that are appropriate for the current view or selection. The menu that is displayed is referred to as the *<Operate> menu*.
<Window> button	Bring up the menu of actions that can be performed on any VisualWorks *window* (except dialogs), such as **move** and **close**. The menu that is displayed is referred to as the *<Window> menu*.

These buttons correspond to the following mouse buttons or combinations:

	3-Button	2-Button	1-Button
<Select>	Left button	Left button	Button
<Operate>	Middle button	Right button	<Option>+<Select>
<Window>	Right button	<Ctrl> + <Select>	<Command>+<Select>

Note: On Windows, a two-button mouse may give the wrong menu if a three-button mouse driver is installed. If you have this problem, load the JuggleButtons parcel in the **`goodies`** directory, which fixes the problem.

Getting Help

VisualWorks Non-Commercial is provided as is without any technical support from ObjectShare. There are on-line sources of help and a number of good books.

There is a web page for VisualWorks Non-Commercial at http://www.objectshare.com. This provides up to date information on the release.

The University of Illinois at Urbana-Champaign have kindly provided a mailing list for users of VisualWorks Non-Commercial. To (un)subscribe, you need to send a message to: **vwnc-request@cs.uiuc.edu** with the SUBJECT of *subscribe* or *unsubscribe*.

Finally there are a number of good books on Smalltalk in general and VisualWorks in particular. Two books which concentrate on the VisualWorks (Version 2.5) are:

- *Smalltalk: An introduction to application development using VisualWorks*
  Trevor Hopkins and Bernard Horan
  Prentice-Hall, ISBN 0-13-318387-4
- *Smalltalk by Example. The developer's guide*
  Alec Sharp
  McGraw-Hill, ISBN 0-07-913036-4

(page deleted; not applicable to the non-commercial version)

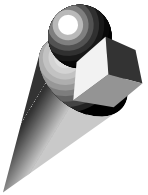
Additional Sources of Information

In addition to this manual, VisualWorks includes a number of other documents. The ObjectShare publications website (http://www.objectshare.com/doc) has VisualWorks documents that can be viewed or downloaded.

Online Cookbook

The *VisualWorks Cookbook* is available online for quick reference to common procedures. Before accessing the online Cookbook, you must load the Help parcel, **`help.pcl`**.

To display the online documentation browser, open the **Help** pull-down menu from the VisualWorks main menu bar and select **Open Online Documentation**, then select the Cookbook.



Chapter 1

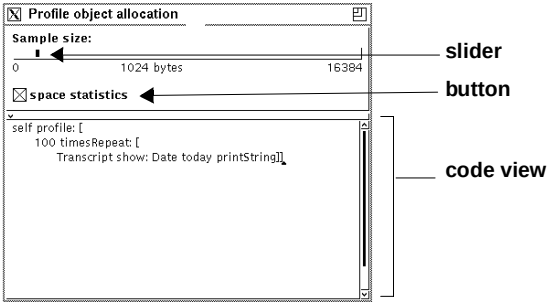
Profiling Time and Memory Usage

The Time Profiler helps you locate portions of your code that consume undue amounts of processing time. The Allocation Profiler performs a similar service for memory usage.

The user interface is very similar for both profilers, so they are often discussed generically in this chapter — "profiler" refers to both equally.

Creating an Object Allocation Profiler

To open a Time Profiler, select **Launcher's Tools→Advanced Menu→Profiles→Time**. To open an Allocation Profiler, select **allocations** in the submenu. A profiler window contains the following components: a code view for entering the code to be analyzed, a slider control for adjusting the sample size and, in the Allocation Profiler only, a **space statistics** button to extend the coverage of the analysis. Each of these components is discussed further below.



The parts of a profiler

Profiling a Block of Code

To create a profile of time or memory usage, enter the Smalltalk expressions in the code view of the profiler encased in a self profile: [ ] expression (an example is provided as a template when you open a new profiler). For example, suppose you wanted to find out what proportion of the memory allocated by the following expression was allocated by the Date method:

```
self profile: [Transcript show: Date today printString]
```

Enter the expression in the code view of an Allocation Profiler, then highlight it and select **do it** in the <Operate> menu. After the expression is executed (today's date is printed in the System Transcript), the results of the analysis are displayed in a new window. For an explanation of the report, see "Analyzing the Object Allocation Profile" on page 4.

In the Allocation Profiler, click on the **space statistics** check box to include a summary of object/byte allocations by class. This summary is described on page 10.

Optimizing the Sample Size

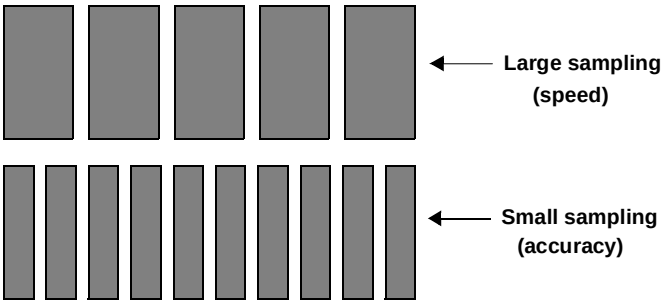
A profiler typically provides only an approximation of the time or memory being used by each method that is called. It does so, in effect, by monitoring the process at a regular interval, called the *sampling interval*. For example, if a babysitter checks in on children playing in their room every half hour, the sampling interval is 30 minutes.

At each 30 minute check point, the babysitter has to assume that the behavior of the moment has been going on for the past half hour. By reducing the sample size to 15 minutes, the babysitter will get a more accurate picture of the children's activities, though it will cost more time and effort.

The sample size can affect the accuracy of the results dramatically. Reducing the sample size improves the accuracy, but may slow down the profiling run disproportionately. Setting the sample size

to zero, for example, causes the profile to be updated after each indivisible chunk of time or memory is used, which can be very time-consuming. In most situations, the default sample size provides adequate accuracy without slowing things down unnecessarily.

To reduce the sample size (for brief processes), move the slider to the left until the desired numerical size is shown below the slider. To increase the sample size (for time- or memory-intensive processes), move the slider to the right. (To move the slider, place the cursor on the dark bar, press and hold the <Select> button on the mouse, then move the mouse to position the slider.)



Speed versus accuracy trade-off when adjusting the sample size

In the example used above, printing today's date in the transcript, the process is so light in its memory usage that the default sampling interval of 1024 bytes is inappropriate. The process is only monitored a few times, resulting in misleading allocation statistics. The obvious solution is to reduce the sample size so the process is checked more frequently.

An alternative technique is to leave the sample size at the default, but repeat the process many times. We can accomplish this by entering the following expression in the code view of the profiler:

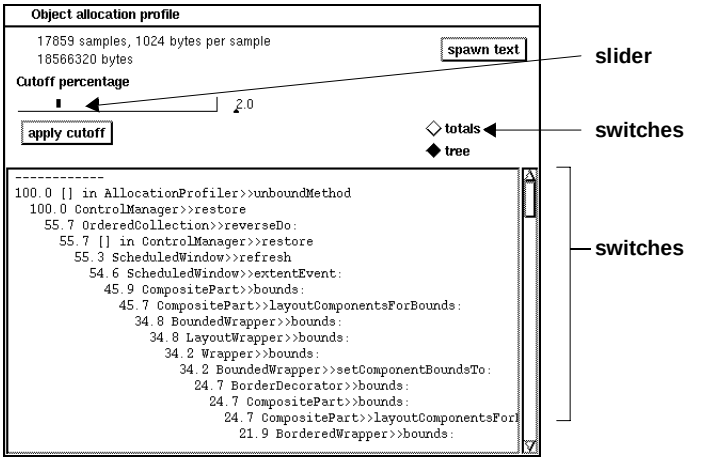
```
self profile: [100 timesRepeat: [Transcript show: Date today
 printString]]
```

This approach often gives superior accuracy because the odds of one or more checkpoints occurring in a low-consumption part of the process are improved. In our example, it turns out that the Date today part of the process only allocates about 3 percent of the bytes, so in a single pass it would be overlooked unless the sample size was very small.

Analyzing the Object Allocation Profile

After the process that you are profiling has finished executing, the profile is displayed in a profile window having the following components:

- A record of the sampling parameters.
- A slider for changing the cutoff percentage and a button for applying a new percentage.
- A text view for displaying the statistics.
- A **totals** switch and a **tree** switch for selecting the type of statistics to be displayed in the text view; and, in an allocation profile for which the **space statistics** check box was turned on, a third switch labeled **space usage** for displaying those statistics.



The structure of a profile window

Each of these components is discussed further below.

At the top of the profile window, a set of statistics display useful information about the profiling run, which include:

- Number of samples
- Sample size
- Total bytes consumed (allocation profile)
- Total milliseconds consumed, in both elapsed and processor time (time profile).

This information is useful in judging whether a change in the sampling interval will prove fruitful—because relatively few samples were taken, for example. This information also serves to label the profile, distinguishing it from profiles generated by other sampling runs on the same code.

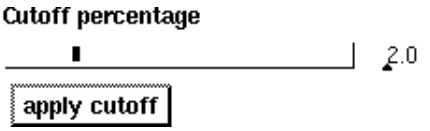
Tree Report

When the **tree** switch is selected, the text view displays a listing of consuming methods that were called during the process. This listing is useful for locating the places in your code that consume the most time or memory, and therefore merit your optimizing attention.

Each method selector is preceded by a number representing the percentage of system resource (bytes or milliseconds) consumed by that method. The tree is displayed in the form of an indented list—each method is indented under its calling method.

Adjusting the Cutoff Percentage

Only those methods that consumed more than a threshold percentage of time or memory are shown. The default is 2 percent, meaning any method that consumed less than 2 percent of the time or memory is excluded from the listing. In effect: “If it’s smaller than this, don’t bother me with it.”

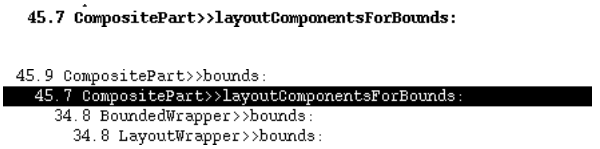


The slider and button used to change cutoff percentage

To get finer detail in the profile, reduce the cutoff percentage by moving the slider to the left. To restrict the profile to the methods that consumed larger chunks of time or memory, move the slider to the right. After you have changed the position of the slider, apply the new cutoff by clicking on the **apply cutoff** button.

Contracting and Expanding the List

Another means of making the list more manageable in size is to temporarily remove selected subhierarchies from the display. To do so, select the method above an unwanted subhierarchy and then use the **contract fully** command. The selected method redisplays in boldface, indicating that it can be expanded to show more detail; its called methods will be eliminated from the display.

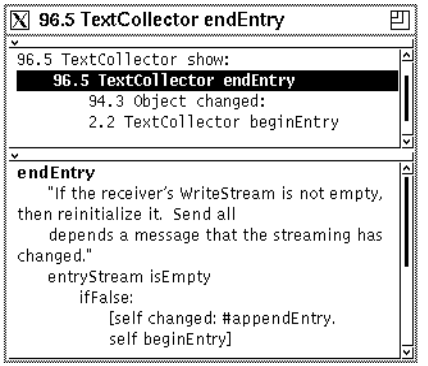


A profile entry contracted and expanded

To restore detail under a contracted method, use either **expand** (for a single level of called methods) or **expand fully** (for the entire subhierarchy) in the <Operate> menu.

Spawning a Method Browser

To examine the body of a method in the tree, select the desired method and then use **spawn** in the <Operate> menu. A method browser will be opened in a separate window. Besides the selected method, which is listed in boldface in the new window, the browser will list parent and child methods when appropriate.



A Method Browser on the selected method and its neighbors

While the browser offers most of the features of a code view, including text editing, you cannot recompile an edited method (via **accept**) in this window because that could cause confusion about the state of the code at the time of the profile.

You can also browse **senders** of the selected message, **implementors** of the method, and implementors of **messages** contained in the selected method. These operations are the same as in the System Browser.

Totals Report

When the **totals** switch is selected, the text view displays a list of the fundamental object-creating methods that were called, with the percentage of system resource consumed by each.

```
13.6 Object copy
11.6 Point +
9.8 Point -
8.5 ColorValue luminance
8.0 Rectangle class origin:corner:
7.6 LuminanceBasedColorPolicy renderPaint:usingPalette:
4.5 Point class x:y:
3.6 GraphicsContext clippingRectangleOrNil
2.7 Rectangle class origin:extent:
2.2 GraphicsContext translation
```

A sample "totals" report

For example, a process that deals with graphics might make many calls to the x:y: method in the Point class. That activity would be summarized here. If you felt Point was taking an inordinate amount of time or memory to get the job done, you might investigate alternative coding paths that would generate fewer messages to Point.

To open a code browser on a selected method and its surrounding contexts, use the **spawn** command as described above.

Space Usage Report

When the **space usage** switch (only available in an allocation profile) is selected, the text view displays a list of object types that were created—technically, a list of classes that were instantiated. For each, the number of instances is indicated along with the cumulative memory usage (in bytes). The cutoff percentage has no effect on this report—all classes that allocated objects are listed.

This report differs from the **totals** report in two important ways. First, it summarizes the activity by class rather than by object-allocating methods within classes. For example, Point>>asPoint and Point>>+ might be listed separately in the **totals** report but they are subsumed under a single entry for Point in the **space usage** report.

Second, **space usage** shows the number of instances and the *amount* of memory used, while **total** shows a *percentage* of allocated memory.

```
    Class                       Instances       Bytes
Point                             6563          131260
Rectangle                         2083          41660
Float                             1694          27104
Interval                          181           4344
GraphicsContext                   156           13104
Fraction                          92            1840
DisplayScanner                    57            5472
ByteString                        50            1183
Array                             36            584
FontDescription                   30            1200
CharacterBlock                    28            784
CharacterBlockScanner             27            3132
RunArray                          18            504
ColorValue                        8             192
Text                              6             120

  Allocation summary:
11029 total objects, average size 21.1 bytes.
1744 byte objects, average size 16.2 bytes.
9285 pointer objects, average size 22.0 bytes.
```

A sample "space usage" report

An allocation summary is provided at the bottom of the **space usage** report, which shows a count of total objects and the average size of each object. This information is broken down by byte-type and pointer-type objects.

Overview of the Code

The following classes provide the kernel of profiler functionality:

- Profiler and its subclasses, TimeProfiler and AllocationProfiler
- MessageTally, a subclass of Magnitude

In early releases of VisualWorks, the MessageTally class provided time-profiling behavior in addition to its current reduced role. The profiling part of its functionality has been factored into Profiler, which provides more general support for assessing usage of an arbitrary system resource. The two subclasses, TimeProfiler and AllocationProfiler, specialize that spying ability for specific resources.

This architecture aligns with the fundamental notion that any system resource can be metered with the sampling apparatus provided by Profiler and the storage mechanism provided by MessageTally. For example, you could construct new subclasses of Profiler to measure disk seeks.

The newly trimmed-down version of MessageTally represents a single node in the tree-like hierarchy of message-sends that occur during the process being profiled. When the profiler stops the running process to take a sample, a MessageTally is created for the method that is currently executing (unless that method was already tallied in the previous sample, in which case its tally is simply updated). Then instances of MessageTally are created and/or updated for the calling methods.

Each MessageTally remembers its place in the calling tree by holding onto its caller and its callees. This permits the report generator to construct the indented list known as the **tree** report.

Allocation Profiler's Wrapped Methods

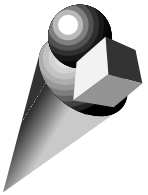
While TimeProfiler enforces its sampling interval straightforwardly, by monitoring the system clock, AllocationProfiler requires a more complicated mechanism. It maintains a list of primitive methods that allocate space for objects. Each time one of these methods is called, the original method is renamed. In its place, a "wrapped" version is substituted. This new version meters the memory usage in addition to performing its original function. At the end of the profiling run, AllocationProfiler restores all such wrapped methods to their original state.

AllocationProfiler assembles its list of allocating primitives during initialization of the class. The list and the resulting cache can become out of date when you add, delete or change a primitive method. Before using AllocationProfiler after you have filed in or otherwise recompiled code containing primitive calls, execute AllocationProfiler initialize.

Time and Space Overhead

The profilers impose relatively minor time and space overhead on the running process. Time overhead depends on the sampling frequency you choose—with the default of 16 milliseconds, the process will take roughly 50–70 percent more time than in its unmonitored condition. Memory overhead varies depending on the nature of the code.

The classes described in this chapter are useful in applications requiring advanced mathematical constructs such as complex numbers and infinity.



Chapter 2

Class Reports

Overview

The Class Reports tool performs a variety of automated checks on specified classes and helps you:

- Repair common coding errors.
- Estimate memory requirements of your application.
- Document your code.

Class Reports is a specific tool that is built on top of a set of general system-analysis capabilities. Those system-analysis facilities could well be put to use in other ways as well.

Creating Class Reports

To open a Class Reports window, select **Launcher's Tools→Advanced Menu→Class Reports**.

The Class Reports window contains the following components for defining the contents of the report:

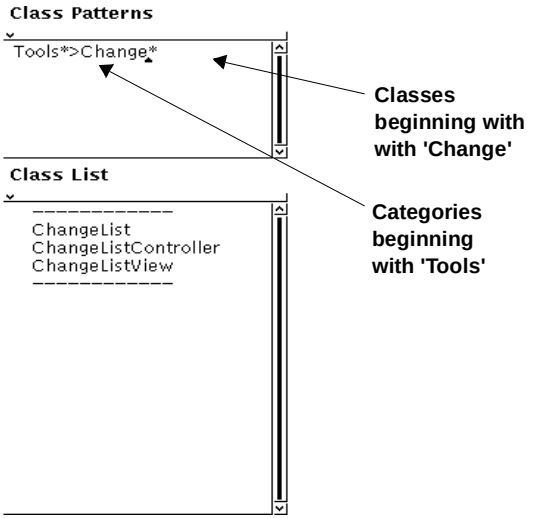
- A Class Patterns view for roughly defining the classes to be checked.
- A Class List view for selecting individual target classes.
- Three switches for choosing a type of report.
- Depending on the type of report selected, two extra switches may be provided for choosing the output destination.

- Depending on the type of report and the output destination, additional options may be provided.
- A button labeled **run** for launching a scan-and-report sequence.

Selecting the Target Classes

You can generate a report for a single class, all classes or any list of classes. Keep in mind as you assemble your list that the amount of time required to produce a report increases with each added class.

Use the Class Patterns view to make a rough cut at the list. Enter one or more wildcard patterns, one per line. Each such entry can contain a class category component and/or a class component. If both components are present, separate them with a greater-than symbol (**>**). Then choose **accept** in the <operate> menu to display all classes matching those criteria in the Class List view. Wildcard patterns are not case sensitive; an asterisk (*****) stands for any string, and a number sign (**#**) stands for any single character. You can also use the **paste** command to insert a list of patterns that you use frequently.



Using a wildcard pattern to define a work list of classes

The following examples are all valid class patterns:

Valid class patterns

Tools*	Classes in categories beginning with 'Tools'
tools*	Same as above
Tools-Misc>*	Classes in the Tools-Misc category
Tools*>Changes*	Classes beginning with 'Changes' in categories beginning with 'Tools'
Changes*	Classes beginning with 'Changes'
ChangesList	The class name ChangeList

Then, in the Class List view, click on the desired class or classes to highlight them for inclusion in the report. Use the **add all** command in the <operate> menu to select all of the classes in the list at once; use **clear all** to deselect all of them. To select a range of classes, hold down the <Shift> key while dragging through the desired class names; to deselect a range of classes, hold down the <Control> key while dragging.

Locating Coding Errors

To scan the selected classes for coding errors, select the **Correctness** switch in the upper left corner of the Class Reports window. Two new switches will appear, labeled **Report** and **Browse**. When the **Report** switch is selected, ten report options are displayed. Each option has a check box, and you can check any number of them to build up the desired report. When the **Browse** switch is selected, eight of the options are offered—the other two are only appropriate for report output.

Class Report Options

Messages Sent but Not Implemented

Each method in the class is checked to make sure that every message sent is implemented somewhere in the system. No attempt is made to assure the appropriateness of the implementor. For example, a self grok message is acceptable even if grok's implementor is not in the target class or its superclass hierarchy.

Methods that send an unimplemented message are reported or, in **Browse** mode, listed in a browser for examination and possible correction.

Messages Implemented but Not Sent

Each method in the class is checked to make sure that its selector is sent by at least one calling method.

Defining what it means for a message to be "sent" is problematic. As an extreme example, one could have code that says self perform: (a,b) asSymbol, where a and b are variables that hold 'foo' and 'bar', respectively. This code, then, sends the message foobar, but no practical analyzer can figure this out. So system tools have to take a particular stand as to what it means for a message to be sent.

In the case of this facility, the stance taken is exactly the same as that taken by the **senders** and **messages** facilities in the System Browser: a message is sent if some compiled code has the message selector as a literal. It will be a literal if the selector is used in code (e.g., self foobar), or if the symbol exists in literal form (e.g., self perform: #foobar). (The exception to this rule is a set of special selectors known by the compiler classes. These selectors are always considered to be sent, even if they do not appear as literals anywhere.)

As a result, the facility may falsely report that some implemented messages are not sent, so the report should be used as a guide. The above example is, of course, poor programming style.

Methods that are not sent are reported or, in **Browse** mode, listed in a browser for examination.

Method Consistency

When two messages sent to the same instance or class variable assume different object types, a conflict is reported.

Similarly, when a temporary variable is used to hold two very different kinds of objects (considered bad form) and thus is sent incompatible messages, a conflict is reported.

The current value of each class variable, pool variable and global variable is also tested to be sure its class implements the messages that are sent to it.

Finally, an inconsistency is reported when a message is sent to self that is not understood by the self object.

When inconsistent methods are found, they are reported or, in **Browse** mode, listed in a browser.

Subclass Responsibilities Not Implemented

Each method that consists of a self subclassResponsibility message motivates a check of each leaf subclass to make sure it owns or inherits a reimplementation of that message.

Note that abstract subclasses need not implement these messages—in such cases, the report will falsely report errors, so use the report as a guide.

Offending methods are reported or, in **Browse** mode, listed in a browser.

Undeclared References

Each method in the class is checked to verify that no undeclared literals are used. Offending methods are reported or, in **Browse** mode, listed in a browser.

Instance Variables Not Referenced

Each instance variable is checked to make sure it is referenced by at least one method. Unreferenced variables are reported; this option is not available in **Browse** mode.

Check Comment

The class comment is checked to make sure it mentions all instance variables, class variables and class instance variables that are in the class definition.

The comment is expected to follow a particular syntax:

- Any amount of plain text followed by a line that says “Instance Variables:”.
- After that line, there should be a line for each instance variable, containing the variable’s name followed by one or more spaces and tabs, followed by a “type” specification in angle brackets, followed by one or more tabs and spaces, followed by text describing the variable.
- If the class has indexed instance variables, include another line as described above, substituting “(indexed instance variables)” for the variable name.

The type specification is typically one or more class names, or nil, separated by vertical bars. In place of class name, you can also use "ClassName of: OtherClassName", for example "Array of: Boolean". The syntax allows more complicated descriptions; for more information, see the method comments in Parser>>typeExpression and Parser>>simpleType.

If the class defines any class variables, the comment should have a section similar to the instance variable section. The heading line is expected to say "Class Variables:".

Finally, if the class has messages defined as self subclassResponsibility, these messages should be listed in a section with "Subclasses must implement the following messages:" as its heading.

The parsing of class comments is somewhat rigid and sometimes what appears to be a valid comment will generate errors in this report, so use the report as a guide. For example, if a type description does not fit on one line, or if the variable description does not start on the same line, the facility will report these as errors.

For instance variables, the facility performs a protocol test:

- All messages sent to each instance variable are verified as being implemented for the named class (or, if more than one class is named, for at least one of them).
- If the class has existing instances, each variable is expected to hold an object of the named type.
- For each class variable, the current value is expected to be an object of the named type.

This option is not available in **Browse** mode. If a comment contains the words UNDER DEVELOPMENT (in capital letters), that fact is reported and no checking takes place for that class.

Backward Compatibility Message Sends

The methods are checked to see whether they send messages that exist (only) in a backward compatibility protocol.

Indefinite Backward Compatibility Message Sends

Similar to the preceding option, but the checker only pays attention to the ambiguous case, when a message send exists in both a backward compatibility category and another category. In this situation, static analysis cannot determine whether the message send is inappropriate, so it is reported as a candidate for your further investigation.

Backward Compatibility Class References

The methods are checked to see whether they refer to a class that is in a class category that contains the string 'backward compat' (without case sensitivity).

Estimating Memory Requirements

To receive an estimate of the memory requirements of the target classes, select the **Space** switch in the upper-right portion of the Class Reports window. Three new switches will appear. Each button provides a different perspective on the estimated memory requirements, as follows:

- Class Size—For each target class, the report shows the estimated number of bytes required for the class definition, variables, methods and class organization.
- Method Size—For each method in a target class, the following measurements are reported:
  - **Code Bytes**—the memory occupied by the method's byte code, the portable compiled form of the method that is used to create native machine code.
  - **Literals**—the number of literal pointers created by the compiler to refer to such things as message selectors, arrays, strings and floats. Each such literal pointer contributes 4 bytes to the total.
  - **Literal Bytes**—the number of bytes required by literal objects other than Symbols.

- **Full Blocks**—the number of full blocks in each method. Full blocks are blocks that contain out-of-scope references to temps, or nonlocal (^) returns. Full blocks are nonoptimal because they are slower and use more dynamic memory. This is only of concern in methods that are used frequently.
- **Total**—the estimated total number of bytes needed by each method, including overhead (20 bytes) not reported in the other columns. A total byte count for all methods is also displayed.

- Instance Size—For each target class, the following measurements are reported:
  - **Count**—the number of instances that exist.
  - **TotBytes**—the memory, in bytes, occupied by all instances.
  - **AveByte**—the average number of bytes for each instance.

  A summary line reports the same measurements for all target classes.

  These reports are intended to help you optimize memory usage by identifying places in your code where memory usage is disproportionate to the functional contribution of the code.

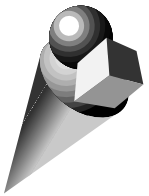
Documenting Your Code

To create a listing of some or all of the elements that make up the code in the target classes, select the **Manual** switch in the upper left portion of the Class Reports window. Two new switches will appear, labeled **Report** and **Print**. When the **Report** switch is selected, the documentation is displayed in a separate window. When **Print** is selected, the output is sent to a printer instead.

The following check-box options are provided for defining the code components to be included in the listing. The options are hierarchic and interconnected, as follows:

- **class definition**
  - **class comment**
- **include metaclass**—include the metaclass definition.
- **protocol names**—instance protocol names are reported; class protocol names are included when the **include metaclass** check-box is selected.
- **include private protocols**—include any protocol beginning with the string "private." Private protocols are made separable in this way because only public protocol is relevant for certain types of manuals.
- **methods**—list method selectors, including metaclass and private methods if those check-boxes are selected.
  - **method comments**
  - **method bodies**—including method comments.

Various text emphases are used for the different components of documentation. For example, #italic is used for the class comment. To change one of these emphases, modify and recompile the appropriate method in the emphases protocol on the instance side of the ManualWriter class.



Chapter 3

Full Protocol Browser

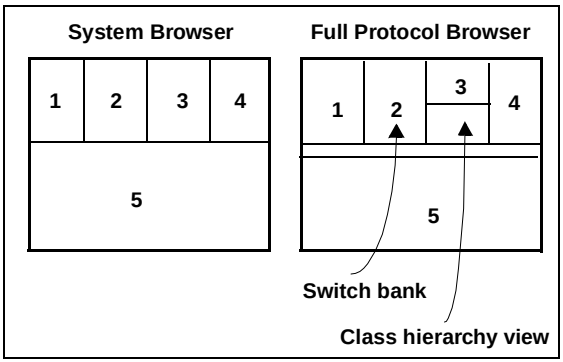
The Full Protocol Browser is an expanded version of the System Browser. It has all of the capabilities of a standard System Browser. In addition, it enables you to include superclass and subclass protocol in the message category and message views. You can also filter the methods by class.

This hierarchic view of a class's functionality can be especially helpful under the following circumstances:

- When you are exploring unfamiliar code, because the Full Browser presents the full behavior set of each class.
- When you are modifying a polymorphic method, because the Full Browser makes it easy to trace inherited behavior.

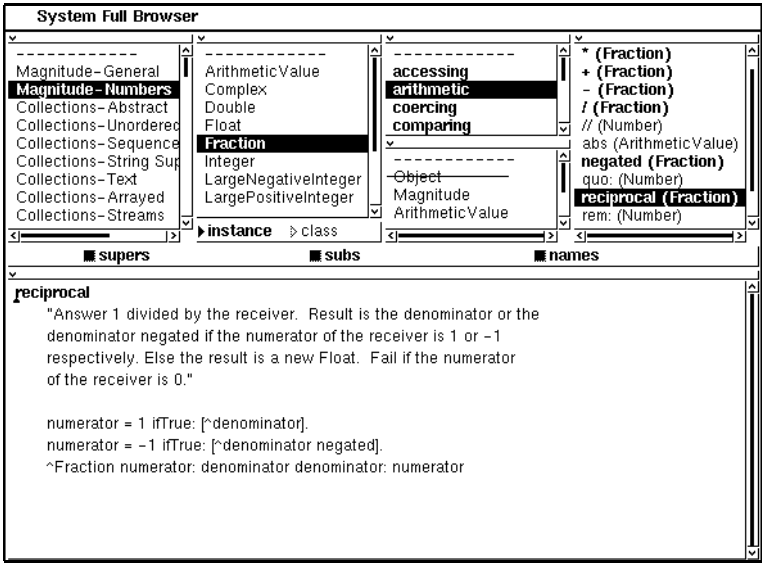
Creating a Full Browser

To create a Full Protocol Browser, select **Launcher's Tool→Advanced Menu→Full Browser**.



System Browser compared to Full Protocol Browser

A Full Browser appears much like a standard System Browser, with the addition of a class hierarchy view, as shown in the following figure. In addition, three switches are provided for filtering the browser's display.



A Full Browser, with the ArithmeticValue *class selected*

The class hierarchy view enables you to filter out parts of the hierarchy and to perform cross-reference searches that are limited to the hierarchy, as described in later sections.

Displaying the Full Protocol of a Class

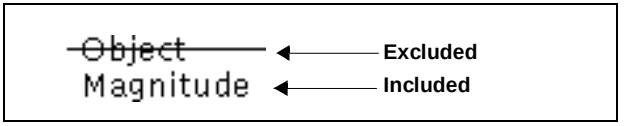
As shown in the above figure, selecting a class such as Fraction in the class view causes the class's hierarchy to be displayed in the hierarchy view. The current class displays in boldface type as a visual cue.

All messages and message categories in this hierarchy display in the appropriate views. The message category view, also known as the protocol view, differs from a view in the System Browser in that the entries list alphabetically. In the message view, polymorphic messages are repeated unless you filter them out, so each method selector can be identified by the class in which it is implemented. In the above figure, for example, the reciprocal method is listed twice, once for ArithmeticValue and again for Fraction. Messages in the current class are displayed in boldface for visibility.

By default, the Object class is excluded from the active list so it is displayed with a line through it. The following section tells how to include and exclude classes from the list.

Filtering Messages by Class

The hierarchy view enables you to filter unwanted classes from the protocol views. To exclude a class, click on it in the hierarchy view. It will be redisplayed with a line through it. To exclude multiple classes that are listed in sequence, hold down a <Shift> key while dragging through the classes to be excluded.



The appearance of included and excluded classes in the hierarchy view

To include a class that was previously excluded, click on it. It will be redisplayed without the line through it.

To include multiple classes that are listed in sequence, hold down the <Control> key while dragging through the classes to be included.

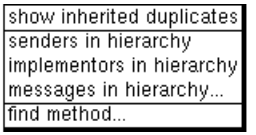
Use the switches in the switch bank to set up default filtering that suits your purposes. Two of the switches provide a convenient means of including or excluding protocol for all superclasses except Object (supers), or all subclasses (subs). By default, duplicate inherited methods are not shown (because they are overridden by the local method)—to show them, select **show inherited duplicates** in the hierarchy view's menu.

The third switch, **names**, toggles whether the implementing class is identified after each method selector.

Searching within the Hierarchy

The **senders** command in the message view's menu operates as it does in a System Browser, searching all classes in the system for methods that se

To limit the search to methods implemented by a class in the current hierarchy, select **senders in hierarchy** in the <Operate> menu of the hierarchy view. Note that all classes in the hierarchy are included in the search, regardless of whether they are filtered out of the protocol and message listings. This is typically much faster than a search of the entire library, and tends to exclude uninteresting implementations. Similar hierarchic counterparts for the **implementors** and **messages** commands are also available in the hierarchy view's menu.



The <Operate> menu of the hierarchy view, used to limit scope of search

Scoping Rules

The hierarchy view's menu also offers a **find method** command, which differs from the protocol view's command of the same name in two ways. First, because the list of selectors may be very large, you get an opportunity to filter it by specifying a wildcard pattern. Second, the implementing class is shown for each selector, and duplicates are listed in inheritance order.

The scope of the commands in the class view's menu and the protocol view's menu are limited to a single class, as in a standard System Browser. However, when a method selector is highlighted, the commands relate to that class. Otherwise, they relate to the class that is highlighted in the class view. (In **FullBrowser**'s code, **selectedClass** and **nonMetaClass** refer to the method view's class, while **currentClass** and **currentNonMetaClass** refer to the class view's class.)

For example, suppose you have selected ArithmeticValue in the class view and then you highlight the denominator (Fraction) entry in the message list view. When you select the **comment** command to display the class comment, Fraction's comment is displayed. To see the comment for ArithmeticValue, select a message for that class (or no message at all).

To restate this scoping mechanism, the selected message's class overrides the class view's class.

There are two exceptions to this rule: the **remove** and **rename as** commands in the message category view. Removing or renaming a message category affects the class that is highlighted in the class view, in all circumstances.

The scope of a message category is extended in a perhaps unexpected but useful way in a Full Browser. As you would expect, when you select a message category such as comparing, all comparing methods in the filtered hierarchy are listed. In addition, methods in superclasses and subclasses that have the same selectors as comparing methods in the current class are included, even if

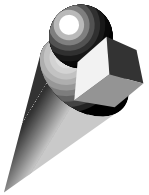
they are located in protocols other than comparing. In other words, when the same selector appears in two different protocols in the hierarchy, Full Browser lists those that could conceivably be grouped in the current protocol because they match qualifying selectors in the current class.

For example, suppose you select the accessing protocol for the Integer class. Both Integer and its subclass LargePositiveInteger implement a method called digitLength. The LargePositiveInteger version of digitLength would be included even if it were housed in a protocol named other than accessing. This behavior obeys the convention that polymorphic messages are placed in protocols of the same name, while allowing for human error and personal choice in the enforcement of that convention.

In summary, the changes in the scoping rules compared with a standard System Browser are as follows:

- Class and protocol view commands apply to the class of the selected message, if any; otherwise, they apply to the current class. Exceptions are the **remove** and **rename as** commands, which always apply to the current class.
- In the hierarchy view, the **find method** command applies to the filtered hierarchy while the other commands ignore the filters.

Conflicts in protocol names for polymorphic messages are ignored.The Time Profiler helps you locate portions of your code that consume undue amounts of processing time. The Allocation Profiler performs a similar service for memory usage.



Chapter 4

Parser Compiler

Overview

The parser compiler classes make it easier to write compilers in Smalltalk. The SQL classes provide an example of an SQL compiler written using the parser compiler facilities.

A typical compiler handles four functions:

- **Scanning**—breaking the source code into tokens (words, numbers, operators, etc.).
- **Parsing**—combining tokens into larger structured units.
- **Semantic analysis**—verifying that variables have been declared, performing type checking, etc.
- **Code generation**—producing a program in machine code or other final form. This may occur in several phases if optimization or more than one representation of the output code is involved.

The parser compiler classes provide the following support for these activities:

- **Scanning**—the Smalltalk Scanner, slightly modified.
- **Parsing**—This phase is the primary focus of the Parser Compiler, providing an efficient language for writing your parser.
- **Semantic analysis**—the Parser Compiler makes it fairly easy to mix in semantics during parsing. This helps to generate an error message that points at the right place in the source code.
- **Code generation**—you're on your own. The Parser Compiler itself demonstrates one style of code generation: It generates

Smalltalk source code during parsing. The complexity of most languages prevents being able to combine code generation with parsing.

Scanning Source Code

The scanner defines seven standard types of token:

- word—a variable or unary message selector
- number—integer or floating point
- character
- string
- binary—infix operators such as + and >=
- keyword—a word followed by a colon (see below)
- signedNumber—a number optionally preceded by a minus sign, with no intervening delimiters

There is an eighth standard token type, keywords, for one or more keywords in succession with no intervening delimiters. This produces a single token. Keywords are only recognized specially if your grammar uses the word keyword or keywords, or if your grammar includes any literal keywords. (This is for the benefit of grammars that don't use keywords, but use the colon for other purposes.)

In addition, the scanner makes assumptions about delimiters (blank, tab, end-of-line and new-page), which separate tokens but aren't tokens themselves. It also assumes that the following characters are tokens on their own: # ( ) | [ ] . : = ^ and ;. To change any of these assumptions requires an understanding of the Scanner's mechanics—you have to write your own initScanner method that calls super initScanner and then substitutes the appropriate entries in the typeTable.

Parsing

For the parsing phase, begin by making your parser a subclass of ExternalLanguageParser—SQLCompiler has been provided as an example. If your source language is method-oriented and you want the output of the parser to be executable CompiledMethods, make your parser a subclass of GeneralParser instead.

This gives your class basic parsing functionality. The parser scans source code one character at a time and one token at a time. You must then write production rules describing the various parts of your language. These rules define parsing algorithms, which your parser will use to recognize constructs such as functions and clauses in the source code. The syntax of production rules will be described in a moment.

Each clause or other construct found in the source code must be instantiated as a node in a parse tree. For example, when an SQL clause is recognized in the source code by SQLCompiler, an instance of SQLClause is created. Classes such as SQLClause typically are subclassed from a more general class such as SQLNode.

As an example of this node-creation mechanism, the production rule implemented by SQLCompiler for recognizing an SQL commit statement creates an instance of SQLStatement as follows:

```
EmulationBorderDecorationPolicy unInstallcommitStatement =
    #COMMIT #WORK
            [statement: 'COMMIT WORK']
```

In this example, the word COMMIT followed by WORK in the source causes execution of the block. A statement: message is sent to SQLCompiler, and that method sends an instance creation message to SQLStatement with the 'COMMIT WORK' string as the statement name.

The ultimate output of the parser is an array containing objects such as SQLFunction, which themselves are often composites of smaller language constructs such as SQLClause. This array represents a parse tree that you can use to generate code.

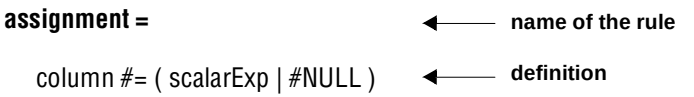
As the parse tree is being assembled, it is stored in an OrderedCollection called stack, held by GeneralParser. This stack responds to collection protocol such as removeLast, and stack operations are frequently embedded in blocks within the production rules. For example, the SQLCompiler>>queryTerm rule contains the following assignment into a temporary variable:

```
tableExp := stack removeLast.
```

A Rule has a Name and a Definition

A production rule describes a semantic unit of the language in terms of other semantic units combined with literal tokens. It introduces the name of the semantic unit, followed by =, followed by the definition, which may include references to other production rules or to literal keywords that are expected at various points in the source-code.

As an example, the following production rule is taken from SQLCompiler:



When a production rule is invoked, its definition is used as a template for the current source code. If the template fits, the rule returns true, triggering creation of the appropriate node in the parse tree. If the definition doesn't match, either the rule returns false, or an error notification occurs.

Rules are Similar to Methods

It is no accident that a production rule looks like a Smalltalk method. It is created just as a Smalltalk method is, by adding it to the instance protocol for your compiler class (SQLCompiler, in this case). You can use the System Browser to do so, or you can file it in. This is possible because the ParserCompiler's responsibility is to take production rules and translate them into equivalent Smalltalk code, which is then translated into an executable method. Each production rule is translated into a method whose selector is the name of the production rule. As a result:

- You can browse production rules in the same way you browse Smalltalk methods.
- Production rules can call Smalltalk code, and vice versa.

Temporary Variables Can be Used

A production rule can have temporary variables. These are defined the same way as in Smalltalk, by enclosing the list of names between two vertical bars.

A production rule begins with a method pattern consisting of the name of the rule, plus names for any arguments. Except for the terminating equal sign (=), the syntax is identical to that of a Smalltalk method, allowing for unary, binary and keyword patterns.

A Rule Definition is a Series of Alternatives

The body of a production rule, called its definition, is a series of *alternatives*, separated by vertical bars (|). The parser tries to match the current source code to each alternative in turn. If a given alternative succeeds, the definition succeeds and returns true. If an alternative fails, the next alternative is tried.

The final alternative in a series can be left empty to return true immediately. If the series is enclosed in parentheses, the empty alternative is indicated by a vertical bar preceding the closing parenthesis. If the series is the body of the definition, the empty alternative is indicated by making a vertical bar the last element of the definition.

For example:

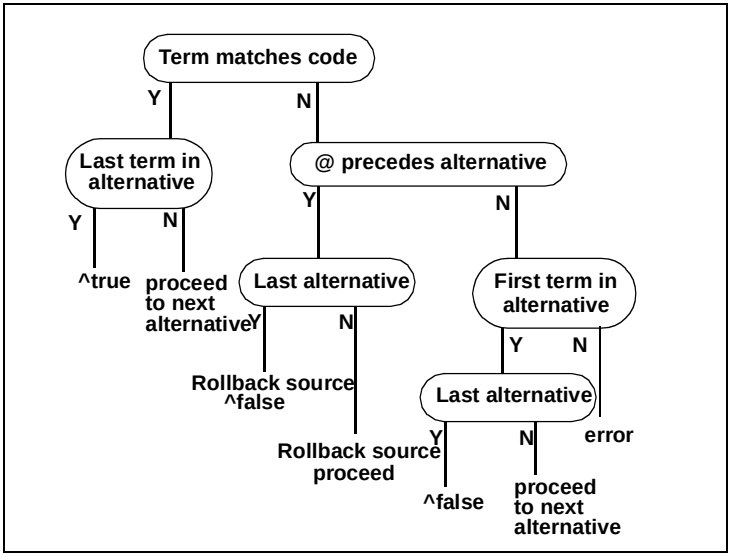
```
( a | b ) c  The next tokens must match either 'a' or 'b', followed by
 'c'

( a | ) c The next token or tokens must match either 'a' followed by
 'c', or 'c' alone
```

An Alternative is a Series of Terms

An alternative is a series of *terms*, each alternative optionally preceded by an at sign (@). Each term is evaluated sequentially against the source code. If a term succeeds, the parser proceeds to the next term; otherwise it fails. If the last term in the alternative succeeds, the alternative returns true. If the alternative fails, behavior depends on several factors:

- If the at sign is present, the source code stream is rolled back to the state it was in when the alternative was started, and false is returned.
- If the term that failed was the first in the alternative, false is returned.
- Otherwise, an error notification is returned.

The following figure summarizes these outcomes in a decision tree showing that action that results when a term is evaluated under various conditions.



Summary of the outcomes in a decision tree

Two examples follow:

```
a b c
```

Expect to find an a, followed by b and c. If a is not found, proceed to the next alternative or return false. If b or c is not found, print an error message.

```
@ a b c
```

Expect to find an a, followed by b and c. If a, b, and c are not found when expected, proceed to the next alternative or return false.

Suppose the parser matches a, but fails to match b. For accurate error detection, the ParserCompiler will not automatically back up on failure, so in this case a message would appear saying b expected. However, it is possible that if the source stream were backed up, we might be able to match c d rather than a b. Therefore, in this case, it is appropriate to write the rule as:

```
@ a b | c d
```

Then, if a succeeds but b fails, the parser will back up and try to match c followed by d.

Another way to think about it is: When the first term in an alternative is matched, the parser assumes it has found the correct alternative. If a later term fails to match, the parser reports an error based on its assumption that the correct template was applied unsuccessfully. The at sign removes that assumption so that, instead of generating an error in this situation, the compiler proceeds to the next alternative.

A Term is an Action or a Unit-Plus-Qualifier

A term can be an *action*, or it can be a *unit* followed by one of the following symbols:

```
*   * ! + +! \   \!  !*
```

We will discuss the more common type of term first: units and their quantifying modifiers.

A Unit is a Word, Terminal or Parenthesized Definition

A unit can be a word, a *terminal*, or a definition wrapped in parentheses. If it is a word, that word is assumed to be the name of another production rule. Some examples:

Word and associated production rule

foo	Evaluate the production rule foo on the current source code. If it returns false, fail the current alternative, else continue.
word=#ABC	If the next token in the source is ABC, push it on the stack and scan another token, else fail the alternative.
keyword=#ABC:	If the next token in the source is ABC:, push it on the stack and scan another token, else fail the alternative.
$(	If the next token is the open parenthesis character, scan another token, else fail the alternative. The stack is unaffected.

Word and associated production rule (Continued)

#ABC	If the next token in the source is ABC, scan another token, else fail the alternative. The stack is unaffected.
#ABC:[keyword type]	If the next token in the source is ABC:, scan another token, else fail the alternative. The stack is unaffected.
#~=	If the next token in the source is ~=, scan another token, else fail the alternative. The stack is unaffected.
#'<<='	If the next token in the source is <<=, scan another token, else fail the alternative. The stack is unaffected.
(...)	When parentheses are encountered, the enclosed part of the rule is parsed according to the rules for definition on page 33.

The following examples illustrate the use of the seven quantifying symbols with units. In these examples, foo pushes a FooNode onto the stack, while foo2 does not affect the stack.

Quantifying symbols

foo *	Expect zero or more repetitions of foo. The top value on the stack will be an Array of FooNodes.
foo *!	Expect zero or more repetitions of foo. The top N values on the stack will be FooNodes, where N is the number of repetitions.
foo +	Expect one or more repetitions of foo. The top value on the stack will be an Array of FooNodes.
foo +!	Expect one or more repetitions of foo. The top N values on the stack will be FooNodes.
foo \ foo2	Expect one or more repetitions of foo, separated by foo2. The top value on the stack will be an Array of FooNodes.

Quantifying symbols (Continued)

foo \! foo2	Expect one or more repetitions of foo, separated by foo2. The top N values on the stack will be FooNodes.
foo !*	Expect one occurrence of foo. Assume that foo leaves an Array on the stack. Pop the Array off the stack and push each of its elements onto the stack.

A Terminal is a Single Token

A terminal is a single token in the language, such as a number, a string, a variable name or a keyword. In the ParserCompiler, the following terminals are recognized:

- A dollar sign ($) followed by a single character, representing a literal character in the source.
- A number sign (#) followed by:
  - A string (any sequence of characters enclosed in single quotes)
  - A word (an alphabetic character followed by alphabetic characters or digits)
  - A keyword (a word followed by a colon)
  - A binary symbol (anything that represents a legal binary operator in Smalltalk, such as //, \\, *, ~~ and ~=)
- The sequence word=#someWord, where someWord is a word as defined above
- The sequence keyword=#someKeyword, where someKeyword is a keyword as defined above

The difference between #someWord and word=#someWord, is that in the former case someWord becomes a reserved word in the language and is always treated specially. In the latter case, someWord does not become a reserved word and is treated specially only when it is preceded by word=.

An Action is a Block or a Special Symbol

An action can be either a Smalltalk block or one of the following special symbols:

Action symbols

Symbol	Description
<	Saves the source position in a local variable (specifically, the temps instance variable in ParserCompiler). Note that only one source position per production rule is saved, so if you overwrite it, the old value is lost.
>	Assumes that the source position was previously saved via <, and that the top value on the stack is a parse node. The parse node is sent a sourcePosition:to: message, with the saved position as the first argument and the current position as the second argument. This implies that your node classes must implement a sourcePosition:to: message when you use this symbol in a production rule.
<<	Pushes the source position onto the stack.
>>	Assumes that the top value on the stack is a parse node, and that the next value is a source position saved by <<. The parse node is sent a sourcePosition: message, with an interval from the saved position to the current position as the argument. The source position is removed from the stack, and the parse node remains the top element.
?	Pops the top value off the stack. If it is true, proceed, otherwise fail the current alternative.
.	Pops the top value off the stack and proceed.

The first four operations are for matching source code positions to parse nodes. The last two are for use with Smalltalk blocks. When a Smalltalk block appears in a production rule, the block is evaluated and the result is pushed onto the stack. If you are interested in the effect of the block but not the returned value, follow the block with a period to get rid of the unwanted value. To

decide whether to continue parsing after a block has been evaluated, follow the block with a question mark to cause the current alternative to proceed or abort depending on the returned value.

Two Types of Block Syntax are Allowed

Two distinct syntaxes are accepted for Smalltalk blocks. One form of syntax is identical to that of normal Smalltalk blocks having zero arguments. The second form is nonstandard and requires further explanation—it has the advantage of very concise coding, with the disadvantage of very restricted syntax.

Like a normal block, this special block is enclosed in square brackets. It consists of exactly one message—the message can be either a binary or keyword message, but not a unary message. The receiver is specially coded:

- If there is no receiver, the message is sent to the parser itself.
- If the message selector is preceded by a colon (:), the top value is popped off the stack and used as the receiver.

Each of the arguments is likewise specially coded:

- If there is no argument, or if the argument is a colon (:), the top value is popped off the stack and used as the argument.
- If the argument is a normal Smalltalk literal (Symbol, String, Number, Array, ByteArray, Character, or nil, true or false), it is used in the ordinary way.
- If the argument is a temporary variable, instance variable, class variable or global variable, it is used in the ordinary way.

For example, the following block sends a copyWith: message to the top value on the stack, with the second value on the stack as argument:

```
[ :copyWith: ]
```

Note that no argument can be the result of a message send.

Summary of Grammar for Parsing Methods

Here is a simplified version of the grammar for parsing methods, written in itself:

```
method = pattern #= temporaries definition

pattern = word | (keyword word)+

temporaries = $| word* $| |d

definition = alternative ($| alternative)*

alternative = ($@ | ) term*

term = unit
            ( (#* | #*!)
            | (#+ | #+!)
            | (#\ | #\!) unit | )

unit = word | character
    | $# (word | keyword | binary | string)
    | $( definition $)
```

Creating your Own Compiler

In preparation for writing programs in your new language, first define a compiler class MyLanguageCompiler, then define a dummy class MyLanguage. Define the following class method for MyLanguage:

```
compilerClass

    ^MyLanguageCompiler
```

Then any methods defined in class MyLanguage or any of its subclasses will compile with MyLanguageCompiler rather than the standard Smalltalk compiler. The example methods in the SQL class are compiled by SQLCompiler in just this way.

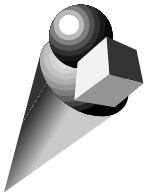
The typical instance creation protocol for a parser takes either a Stream or a String as input, as well as the name of the top-level production rule to be applied. For example:

```
CParser parse: aStream as: #cFile
```

The final step in code generation is done by the message generate:. This message is defined in GeneralParser on the assumption that the output of your compiler (i.e., the single element left on the stack at the end of recognizing a method) is a string that is actually a Smalltalk source method, which then gets handed to the Smalltalk compiler.

However, you can override this method in your own compiler to do something different. It should return a selector if the code generation succeeds, or nil if it fails. In the case of the SQL example, the final object is an Array containing a parse tree in the form of a hierarchy of nodes. Try the examples on the instance side of the SQL class, inspecting the results recursively to see the structure of the parse tree.

This object responds to Smalltalk messages and can thus be manipulated to suit the next phase of compilation.



Chapter 5

Enhanced Numbers

The classes described in this chapter are useful in applications requiring advanced mathematical constructs such as complex numbers and infinity.

Complex Numbers

An instance of class Complex has two components, a real number such as a Float, and an imaginary number (a multiple of *i*, which represents the square root of -1). A Complex number is represented in the following format: (5.5 + 3 i)—white space inside the parentheses is ignored.

Creating an Instance

An instance can be created by using the literal form shown above, or via the real:imaginary: method, as in Complex real: 5.5 imaginary: 3. When the real component is zero, sending the message i to an integer is sufficient, as in 3 i. When the imaginary component is zero, the shorter fromReal: method can be used. In summary, the expressions in the left column generate the Complex numbers in the right column below:

3 i	(0 + 3 i)
5.5 + 3 i	(5.5 + 3 i)
Complex fromReal: 5.5	(5.5 + 0 i)
Complex real: 5.5 imaginary: 3	(5.5 + 3 i)

Protocol Summary

Complex numbers support the usual numeric operations, including accessing, arithmetic, mathematical functions, coercion, comparison, conversion, testing and generality. Nonequal comparison, truncation and rounding are not valid with complex numbers. Additional methods include:

Accessing

r	Same as abs, which returns an absolute magnitude. For example, (5.5 + 3 i) r returns 6.26498.
theta	Return the angle between the receiver and the positive real axis, in radians

Arithmetic

conjugated	Reverse the sign of the imaginary component.

Converting

asPoint	Return a Point with the real component as the x value and the imaginary component as the y value.
i	Multiply the receiver by (-1 sqrt). This message is also understood by Number after **`MetaNum.st`** is filed in.

Metanumbers

MetaNumeric **Class**

Infinity and Infinitesimal are the best examples of metanumbers, which are impossible but mathematically useful constructs. The MetaNumeric class is an abstract superclass with four subclasses, as follows:

MetaNumeric
- Infinity
- Infinitesimal
- NotANumber
- SomeNumber

The MetaNumeric class provides coercion and conversion support for its subclasses. Must of this support comes in the form of double dispatching methods, which bring coercion into play when two unlike numbers fail in some arithmetic or comparison operation.

For example, suppose you execute the following expression:

```
2.3 + (Infinity positive)
```

The Float method for addition doesn't know how to add infinity to a floating point number directly, so it asks the Infinity object to perform the addition. It does so by evaluating:

```
(Infinity positive) sumFromFloat: self
```

The sumFromFloat: method is implemented by MetaNumeric, the abstract superclass of Infinity. After coercing the floating point number into meta form (making it an instance of SomeNumber), the superclass hands off to Infinity to perform the specific addition. All metanumbers need to have non-metanumbers coerced to meta form, so this behavior is performed by their common superclass, MetaNumeric.

Infinity **Class**

Infinity represents a number too large to be represented in any other form. We will use the terms *+infinity* and *-infinity* to denote the positive and negative forms of this number.

It is defined to mean that for any real number *x*, the following is true:

```
-infinity < x < +infinity
```

Creating an Instance of Infinity

The expression Infinity positive creates a positive instance of Infinity, and Infinity negative creates a negative instance.

Protocol Summary

The usual numeric operations are supported by Infinity, according to the following rules (where *x* is any real number):

```
x + +infinity = +infinity
x - +infinity = -infinity
x * +infinity = +infinity when x > 0
x * -infinity = -infinity when x > 0
0 * +infinity = 0
+infinity + +infinity = +infinity
-infinity - +infinity = -infinity
+infinity * (+/-)infinity = (+/-)infinity
-infinity * (+/-)infinity = (-/+)infinity
+infinity - +infinity = undefined value, and an error occurs
```

Because +infinity is not a single value, but a set of all real numbers that are unrepresentably large, it makes no sense to ask whether +infinity = +infinity. Doing this will cause an error.

Infinitesimal **Class**

infinitesimal is a number so close to zero it cannot be represented as a conventional number—it can be thought of as the reciprocal of Infinity.

Creating an Instance of Infinitesimal

Creating an instance of Infinitesimal is done exactly as with Infinity, by executing an expression such as:

```
Infinitesimal positive
Infinitesimal negative
Infinitesimal negative: aBoolean
```

Protocol Summary

We will use the terms *+tiny* and *-tiny* to denote the positive and negative forms of this number.

The usual numeric operations are supported, according to the following rules (where ***x*** is any real number unless otherwise specified):

```
x + +tiny = x when x ~= 0.
0 + +tiny = +tiny
x * +tiny = +tiny when x > 0
x * -tiny = -tiny when x > 0
0 * +tiny = 0
+tiny + +tiny = +tiny
-tiny - +tiny = -tiny
+tiny * (+/-)tiny = (+/-)tiny
-tiny * (+/-)tiny = (-/+)tiny
+tiny - +tiny = undefined value, and an error occurs
x / +infinity = +tiny  when x > 0
x / +tiny = +infinity when x > 0
+tiny * +infinity = undefined value, and an error occurs
```

Loosely speaking, +tiny is not a single value, but a set of all real numbers that are unrepresentably small. As with infinity, it makes no sense to ask whether +tiny = +tiny.

NotANumber **Class**

An instance of NotANumber can be used as a placeholder for the result of an illegal mathematical expression, such as 8 arcSin. Since the behavior of NotANumber consists of various kinds of error signals of the form "You can't do such-and-such with a NaN," the result is substituting one kind of error for another. In theory, NotANumber error signals could be trapped by a signal handler at a high level in your application, which could then decide, for example, to continue with some time-consuming computation, noting the error in a log, rather than abort because of the error. NotANumber was created for the sake of completeness—along with Infinity and Infinitesimal, it is defined by IEEE in the set of floating point numbers.

Creating an Instance of NotANumber

To create an instance, execute NotANumber new.

Protocol Summary

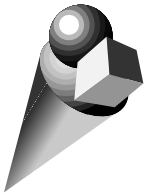
NotANumber implements the common arithmetic and comparison methods, raising an error signal for each.

The printable form of an instance is "NaN" so error strings use that term, as in:

```
'Can't perform arithmetic functions on NaN'
```

SomeNumber Class

SomeNumber represents a conventional scalar number coerced into metanumeric form so it can be used in both conventional and metanumeric computations. Such a number responds to numeric operations as usual, but has the same generality as other metanumbers and can be used in metanumeric computations. It is essentially a support class for the other metanumeric classes so it has little potential for reusability.



Chapter 6

Benchmarks

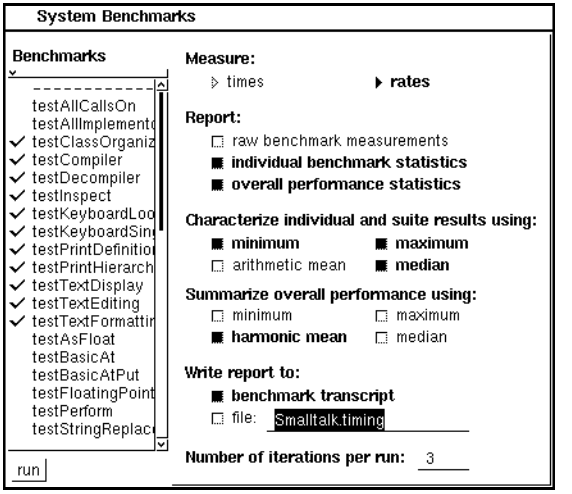
The Benchmark class provides a framework and a convenient interface for running benchmarks to compare your application's performance across versions and in various operating environments. A simple subclass of Benchmark can be built to run the benchmarking tests. As an example, we have provided a subclass called SystemBenchmark, which contains updated versions of the historic test suite we at ObjectShare use to compare system performance on different platforms.

This chapter describes the reusable interface and related mechanisms provided by the Benchmark class, using the SystemBenchmark subclass as an example. The final section then explains how to implement your own benchmarks.

Using the Benchmark Interface

To open the example System Benchmarks window, select **Launcher's Tools→Advanced Menu→Benchmarks**.

In addition to the System Benchmarks window, a Benchmark Transcript window will open to display the test results.



The System Benchmarks window with default settings

The System Benchmarks window has two views, arranged side by side. The benchmarks view, on the left side, lists the available benchmark tests. The parameters view, on the right, contains a variety of buttons and fill-ins for controlling report attributes. A button marked **run** is located below the list view — use the button to begin execution of a test suite.

Assembling the Test Suite

Although a benchmarking run can be limited to a single type of test, such as adding 3 + 4 thousands of times, a run frequently involves a suite of several related tests. You can use the benchmarks view to select the tests you want to include in a run. To select an individual test, just click on it with the <Select> button; click again to deselect it. A check mark appears in the margin next to each selected test.

Selection Techniques

To select multiple adjacent tests, hold down the <Shift> key while dragging the cursor through the desired tests (the check marks will appear after you release both the mouse button and the <Shift> key). To deselect multiple adjacent tests, hold down the <Control> key while dragging through the test names.

To cancel all selections, use **clear selections** in the <Operate> menu; use **select all** to include all of the tests. The subclass can define a default suite of tests— in our example, SystemBenchmark uses as defaults the tests used by ParcPlace for standard comparisons of platform performance. You can reset the test suite to the defaults at any time by selecting **reset to default** in the <Operate> menu. To summarize these operations:

Selection techniques for system benchmarks

Operation	Description
click <select> button	Select and deselect a single test
<Shift> + drag <select>	Select multiple tests
<Control> + drag <select>	Deselect multiple tests
select all	Select all tests
clear selections	Deselect all selected tests
reset to default	Select default tests

Setting the Report's Granularity

At the end of each benchmarking run, a report is generated containing statistics accumulated during the tests. Three buttons at the top of the parameters view control the level of detail in the report, as follows:

Raw Benchmark Measurements

Details about each iteration of each test method. This information can be used to discover significant variations among iterations. The first iteration of an operation, for example, might consume a disproportionate amount of time because it might not take advantage of compiled-code caching.

The following times, for example, were reported for three iterations of two tests in the SystemBenchmark suite: text displaying and text replacement.

```
[display text]                                          First iteration
    10 repetition(s) in
    0.921 seconds
    92100.0 microseconds per repetition

[text replacement and redisplay]
    20 repetition(s) in
    5.1 seconds
    255000.0 microseconds per repetition

[display text]                                          Second iteration
    10 repetition(s) in
    0.88 seconds
    88000.0 microseconds per repetition

[text replacement and redisplay]
    20 repetition(s) in
    4.98 seconds
    249000.0 microseconds per repetition

[display text]                                          Third iteration
    10 repetition(s) in
    0.94 seconds
    94000.0 microseconds per repetition

[text replacement and redisplay]
    20 repetition(s) in
    4.98 seconds
    249000.0 microseconds per repetition
```

Individual Benchmark Statistics

A summary of statistics for each test. In effect, this section of the report summarizes the details described above, whether or not the details themselves are included in the report. This information is useful for identifying the slow performers in a suite of tests, marking them as candidates for optimization.

Results are converted to rates (by the convert:toRateFor: method in the subclass) when the rates switch is selected. When the **times** switch is selected, no such conversion takes place. (The class comment for Benchmark discusses this mechanism and its implications further.) Types of statistics are described in "Choosing Types of Statistics" on page 54.

The following example reports the minimum, maximum and median for the raw times reported in the example above:

Individual benchmark results (three interations)

Benchmark	Minimum	Maximum	Median
TextDisplay	136.170	145.455	138.979
TextEditing	82.7451	84.7389	84.7389

Benchmark Suite Statistics

A summary for the entire suite, the purpose of creating a suite in the first place is to measure the performance of some subsystem. Benchmarking provides a weighted average for the performance of that subsystem, which you can then use to compare with an identical benchmarking run under different operating circumstances.

For the weighted average, the report displays the same columns as for the individual statistics. For example, if you elect to display only the median value for individual benchmarks, only the median value for the suite-wide statistic will be shown.

Benchmark suite results (three iterations)

Rating Type	Minimum	Maximum	Median
Minimum	118.539	126.309	125.558
Maximum	139.13	142.222	142.222
H-Mean	116.364	119.425	118.321
Median	118.539	126.309	125.558

Let's use the minimum H-Mean (harmonic mean) to illustrate the derivation of these statistics further. Each time the test suite is performed, the individual test results are converted to rates and then combined mathematically to arrive at the harmonic mean score for that iteration.

The suite was performed three times, in our example, so three such harmonic means are derived. The minimum H-Mean represents the lowest of the three scores. Similarly, the maximum H-Mean is the highest of the three, and the median H-Mean is the median (or middle value) of the three.

Choosing Types of Statistics

The two summary sections of the report can include different types of statistics. You control which types are included in the report by selecting buttons in the parameters view. The types of statistics are as follows (*i* represents the number of iterations):

- Minimum—the result from the best-performing iteration.
- Maximum—the result from the worst-performing iteration.
- Arithmetic mean—the average of all iterations; sum/*i*.
- Harmonic mean—The number of iterations, divided by the sum of the inverses of the weighted results for the separate iterations.

  $i/[(1/\text{result}_1) + (1/\text{result}_2) + \ldots]$

Note: The median harmonic mean of the SystemBenchmark default test suite is the standard benchmark score used by ParcPlace when comparing system performance in different operating configurations. This test suite differs from the suite used in prior releases of VisualWorks, so the scores cannot be compared across versions meaningfully.

- Median—the value that is midway through a ranked list of the scores. For example, if you specify five iterations, the median is the third element in the sorted collection of scores.

The harmonic mean is only useful when summarizing overall performance, so it is not available under the heading **Characterize individual and suite results using:**. Under the heading **Summarize overall performance using:**, the arithmetic mean is only offered when you select the **times** switch; when the **rates** switch is selected, the harmonic mean is offered.

Setting the Report Destination

The report can be displayed in the Benchmark Transcript window, stored in a disk file, or both. Use the buttons under the heading **Write report to:** in the parameters view to select one or both destinations. You can provide the name of a file in the fill-in blank. The file will be created in the start-up directory unless you specify an absolute or relative pathname.

Setting the Number of Iterations

The test suite can be repeated as a means of improving the accuracy of the results. By default, the iteration count is set to three. To change the number of iterations, type the desired number in the fill-in blank labeled **Number of iterations per run**.

The number of iterations represents the number of times the test suite will be repeated—this is not to be confused with repetitions that are hard-coded into a given method. For example, the

test3plus4 method repeats the 3 + 4 operation 100,000 times for each iteration, so three iterations would cause the operation to be repeated 300,000 times.

In some situations, a single iteration may produce more interesting results. For example, a method might take a relatively long time to execute on its first pass, but run much faster subsequently. However, if your application calls the method only infrequently, the first-iteration results might prove more illuminating.

To begin execution of the testing run, click on the **run** button. If your window manager is configured to prompt you for placement of windows, you might consider turning off that feature before running the default test suite or other suites involving window-displaying operations. However, prompt-for-placement can be left on without affecting the results.

Creating a Benchmark Subclass

The benchmarks are implemented via the following four classes, all of which are subclasses of Object:

- Benchmark, and its subclass SystemBenchmark
- BenchmarkTable
- BenchDecompilerTestClass

Benchmark Superclass

Benchmark is an abstract superclass whose protocol provides the interface we have been describing, as well as the timing and statistical analysis facilities for a benchmarking run. It has instance variables for remembering the report parameters as selected in the interface, and the test results as they are accumulated. Benchmark also provides the reporting protocol, making use of BenchmarkTable (described further below).

SystemBenchmark **Subclass**

Subclasses of Benchmark, such as SystemBenchmark, are responsible for providing the specific tests to be run. See the methods that begin with the word "test" in SystemBenchmark for examples.

In addition, subclasses must implement the following accessing messages:

```
benchmarkLabelForSelector:

benchmarkSelectors
initiallySelectedBenchmarks
```

Subclasses may also need to override Benchmark's weighting protocol, to establish relative weights for test methods and to convert the results to an appropriate rate; and the defaults protocol, which determines the default selections in the user interface.

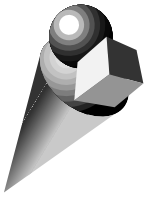
BenchmakTable **Class**

BenchmarkTable provides two-dimensional reporting capabilities that might well be useful to other applications, though the code requires extensions to make it more generally useful. It holds onto a report name, a collection of column labels and a collection of rows. Each row is assumed to be a collection itself.

The protocol is tailored to the needs of the benchmark reports, though it provides a subset of a more generally useful set of behaviors.

BenchDecompilerTestClass **Class**

BenchDecompilerTestClass is a holder for methods that are decompiled during the SystemBenchmark>>testDecompiler benchmark. The code in the methods has no functional value—in fact, it is obsolete.



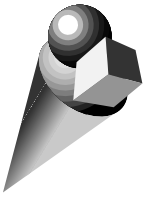
Appendix

Parcels

The following is a list of Advanced Tools parcels and the classes in each.

Category	Filename	Classes
All Advanced Tools	AllAdvancedTool.pcl	no classes; serves to load all other Advanced Tools parcels
AT Benchmarks	ATBenchmarks.pcl	BenchDecompilerTestClass Benchmark SystemBenchmark BenchmarkTable BenchmarkRunner
AT MetaNumerics	ATMetaNumerics.pcl	Complex Infinitesimal Infinity MetaNumeric NotANumber SomeNumber
AT ParserCompiler	ATParserCompiler.pcl	ExternalLanguageParser GeneralParser ParserCompiler PushFragment RecognizerFragment

Category	Filename	Classes
AT Parser Example	ATParserExample.pcl	SQL SQLClause SQLCompiler SQLFunction SQLIdentifier SQLInfixOperation SQLLiteral SQLModifier SQLNode SQLPostModifier SQLStatement
AT Profiling	ATProfiling.pcl	AllocationProfiler MessageTally ProfileMethodListBrowser ProfileOutlineBrowser Profiler ProfilerListHolder TimeProfiler TreeBuilder
AT Support	ATSupport.pcl	DisablingSelectionInList EvaulationHolder GroupingSelectionInList TreeObjectHolder
AT System Analysis	ATSystemAnalysis.pcl	ClassDeclarations ClassNameChooser ClassReporter InstanceTally ManualWriter MessageAnalyzer MessageCollector ReferencePathCollector SystemAnalyzer
AT Tools	ATTools.pcl	FullBrowser OutlineBrowser WindowBrowser



Index